# Implementing Microprocessors from Simplified Descriptions

Nikhil A. Patil
Department of Electrical & Computer Engineering
The University of Texas at Austin
Email: npatil@utexas.edu

Derek Chiou
Department of Electrical & Computer Engineering
The University of Texas at Austin
Email: derek@ece.utexas.edu

***Abstract*—Despite the proliferation of high-level synthesis tools, hardware description of microprocessors remains complex. We argue that much of the incidental complexity can be relieved by untangling the description into separate functional and microarchitectural components. Such an untangling can be achieved using a high-level microcode compiler that can generate not only microcode, but also the micro-instruction format and the interpretations of each control bit. Simplifying hardware description will help the designer make better design-space trade-offs, and close the design and verification loop faster. This paper takes the reader through an implementation of a simple Y86 processor to qualitatively illustrate the complexity reduction from the untangling.**

## I. Introduction

Despite strides in high-level synthesis tools and techniques, implementing processors remains a complex task. Our goal is to simplify hardware description of microprocessors.

To make something fundamentally simpler, one must disentangle interwoven concerns. A braid is more complex than independent strands of rope. Disentangling the strands allows us to study each strand independently and scrutinize it for defects. Processor description entangles the concerns of functional and performance correctness: it is hard to reason whether a certain change to the microarchitecture would preserve functional correctness.

The functional specification of a processor is conveniently specified as a set of instructions operating on a programmer-visible state. Since instruction-sets are usually backward compatible to allow software reuse, they tend to grow over time. The canonical example for this is the x86 ISA, however even ARM and PowerPC, while originally designed for simplicity, have grown to be quite complex. In addition, processors have very tight performance requirements and power budgets that change at each technology node. Thus, over time functional specifications get increasingly divorced from performance specifications, further increasing the entanglement of concerns.

Both the disparity between the ISA and the microarchitecture as well as the sheer size of the instruction set (e.g., there are more than a thousand x86 instructions, considering all variations) make processor implementation even more challenging. We argue that the complexity stemming from ensuring that instructions work as advertised is merely *incidental*, and can be alleviated without affecting the ability to specify *inherently* complex microarchitectural structures like branch predictors, caches, out-of-order issue etc. that enhance performance and save energy. (Note that even if functional correctness can be automatically guaranteed, e.g., by using formal methods, this does not actually simplify the description of hardware itself.)

## II. Microcode Compiler

Imagine that the user could insert "holes" representing combinational logic anywhere in the hardware description of a processor. And imagine that a genie automatically filled these holes, while ensuring that the implementation adhered to the ISA. Such an approach disentangles the instruction set from the microarchitectural specification, since the job of implementing instructions is now delegated to the genie. However, this is a very hard problem, because the genie must be able to reason about the user's code (which might even have bugs), as well as optimize over a large number of axes to come up with a good solution. However, by restricting where these holes can be inserted and by asking the user to specify a functional specification of the description template (implementation with "holes"), it is possible to make this strategy practical.

```
rule RSTG2;                                                    ▷ Execute Stage
    let u1 = stg1.first()
    let x = case(u1.I_ALU_X) matches
                1'h0 : u1.RD1
                1'h1 : u1.RD2
    let y = case(u1.I_ALU_Y) matches
                2'h0 : u1.IMM
                2'h1 : u1.RD1
                2'h2 : 32'h4
    let z = case(u1.I_ALU_K) matches
                2'h0 : x + y
                2'h1 : x − y
                2'h2 : x & y
                2'h3 : x ^ y
    let u2 = ExecUop { IMM: u1.IMM, RD1: u1.RD1,
                       RD2: u1.RD2, ALU: z,   ··· }
    stg1.deq()
    stg2.enq(u2)
```

(a) A Bluespec rule implementing the execute stage

⇓

```
rule RSTG2;                                                    ▷ Execute Stage
    let u1 = stg1.first()
    let x = #alu_x(u1)
    let y = #alu_y(u1)
    let z = case(#alu_k(u1)) matches
                ADD: x + y
                SUB: x − y
                AND: x & y
                XOR: x ^ y
    let u2 = rAdd (u1, ALU, z)                  ▷ extend u1 with field ALU
    stg1.deq()
    stg2.enq(u2)
```

(b) Same code using placeholder functions (*#f*) and extensible records

Fig. 1. Typical Bluespec code for the execute stage in an in-order processor pipeline (condition code generation not shown). Syntax for placeholder functions and record field specifiers has been changed for consistency with the μL language used in the rest of the paper.

Figure 1(a) shows the Bluespec SystemVerilog [1] implementation of the execute stage of a simple processor pipeline. The rule RSTG2 implements an arithmetic/logic unit with four binary operations

(+, −, &, ^). There are three instruction-specific decisions to be made: (a) what is the left input? (b) what is the right input? and (c) which operation to perform? These decisions are encoded into control-bits in the micro-instruction at fields I_ALU_X, I_ALU_Y, and I_ALU_K. It is quite easy to make mistakes in encoding and decoding these bits (in the microcode table and the case statements respectively). We wish to replace such references to control bits by holes that will be filled in automatically. These holes are specified as placeholder functions (*#alu_x*, *#alu_y*, *#alu_k*) in fig. 1(b).

Patil et al. [2] have described a compiler that can generate microcode for an implementation of (i) an instruction set from (ii) the functional description of the connectivity of microarchitectural resources, as well as the micro-instruction format and the interpretation of each control bit. This allows them to implement (iii) the microarchitecture in Bluespec SystemVerilog, in a way that is orthogonal to the instruction set. In other words, they have factored the description so that the user can rapidly swap different instruction sets for the same microarchitecture, and different microarchitectures for the same instruction set. But they have not shown that the three specifications put together are simpler than the complected whole.

The purpose of this paper is to argue that the use of such a microcode compiler does in fact reduce the complexity of describing hardware. This paper showcases μL, a simple domain-specific language, to specify the instruction set and functionality of the microarchitecture template. We have re-implemented the microcode compiler from scratch targeting our new language.

We shall describe our use of this strategy to implement a six-stage pipeline for the Y86 ISA [3] in Bluespec SystemVerilog. Relevant code samples (with few syntactic changes) are included in this paper, and the reader is invited to judge our qualitative conclusions.

## III. Related Work

Our previously published work that describes such a microcode compiler [2] has various awkward restrictions [4]: (a) there is no support for data types or type inference (so the output functions may not be not fully compatible with user-specified Bluespec), (b) compile times can extend to over 30 minutes per instruction for a Y86-like processor, (c) the μop format is very rigid: each resource must insert data into a unique field in the μop, (d) multiple temporary registers are not correctly supported, (e) micro-instructions may generate spurious register-file reads and memory loads (causing pipeline stalls). The current version of the compiler avoids all of these issues.

Sketch [5] is a software synthesis technique that allows "holes" in software implementations to be filled in automatically using functional specifications. An analogous idea has been proposed for hardware synthesis [6]. Gulwani et al. [7] use the Z3 SMT solver to synthesize loop-free programs from functional specifications.

Architecture Description Languages (ADLs) [8] allow the user to specify a behavioral instruction-set description and provide several knobs to configure the generated microarchitecture. For example, Tensilica [9] allows the user to add instructions and data-path elements to the Xtensa processor family. ADLs are also useful to generate instruction-set simulators, compiler back-ends, etc.

C-synthesis languages have rarely been used for processor synthesis. User-guided high-level synthesis [10] is an enhancement over C-synthesis that requires the designer to specify a draft data-path over which instruction behavior is scheduled.

No Instruction Set Computer (NISC) [11] is configured using an XML-based language called Generic Netlist Representation (GNR). The compiler automatically generates a micro-instruction format and compiles software directly to micro-instructions.

| Extensible Records | |
|---|---|
| FIELDNAME | Field specifier of some record |
| *rec*.F | Lookup field F in record *rec* |
| *rec* +: (F, *x*) | Add a new field F to *rec*, changing its type |
| *rec* −: F | Remove field F from *rec*, changing its type |
| *rec* <: (F, *x*) | Update field F in *rec* without changing type |

| Computations over Implicit State | |
|---|---|
| $\Sigma$ | Implicit state record |
| **do**-computation | May involve accesses to implicit state $\Sigma$ |
| $x \leftarrow \cdots$ | Assign value to variable $x$ |
| **compute**$(\cdots)$ | Computation that does not access $\Sigma$ |
| **select**(F) | Lookup field F in record $\Sigma$: i.e., $\Sigma$.F |
| **update**(F, *x*) | Update field F in $\Sigma$: i.e., $\Sigma$ <: (F, *x*) |
| **read**(F, *r*) | ($\Sigma$.F)[r]; i.e., index into array **select**(F) |
| **write**(F, *r*, *x*) | In-place update at index *r* in array **select**(F) |
| **pselect**(F, *p*)<br>**pupdate**(F, *p*, *x*)<br>**pread**(F, *p*, *r*)<br>**pwrite**(F, *p*, *r*, *x*) | *p* - predicated versions of the above |
| **read4**, **write4**, **pread4**, **pwrite4** | Read or write at four adjacent array indexes; these functions perform 32-bit accesses on byte-addressible memories |
| **constrain** | Add a boolean constraint |

| Miscellaneous | |
|---|---|
| *#func*$(\cdots)$ | Placeholder (unknown) function |
| +, −, ∗, | Arithmetic functions |
| &, ^, zeroExtend | Bitwise functions |
| &&, \|\|, not | Boolean functions |
| **if**$(p, a, b)$ | if $p$ then $a$ else $b$ |
| $x$ @: $\tau$ | Type ascription: value $x$ has type $\tau$ |
| Array (Bit $n$) $\tau$ | Type of array of $2^n$ elements of type $\tau$ |

Fig. 2. μL syntax cheat sheet

## IV. Y86

Y86 is a simple 29-instruction ISA used for teaching introductory computer systems [3] in several schools worldwide. We chose a simple ISA so we could talk about more interesting facets of the compiler without complicating the presentation.

The source code consists of: (a) an implementation of the microarchitecture template (figs. 1(b) and 6) in Bluespec SystemVerilog, a high-level synthesis language based on atomic transactions, (b) an ISA description consisting of the architectural state (fig. 4) and a list of instruction definitions (fig. 5), and (c) a functional specification of the microarchitecture template describing the *flow* of a micro-instruction through the hardware (fig. 7). The instruction and flow definitions are in μL, a domain-specific language defined in Haskell. Our overall toolflow is shown in fig. 3

### A. Guide to μL syntax

Figure 2 provides a cheat-sheet for the syntax[1] of the μL language.

*1) Type system:* Like Bluespec, our language for specifying instruction and flow definitions is statically typed; all types must be statically resolved at compile time. The compiler implements classical ML-style type inference, allowing types to be inferred through most

[1]Some minor syntax changes were made for the paper. In particular, the ALLCAPS record field specifiers and *#func* placeholder functions are actually double-quoted strings.

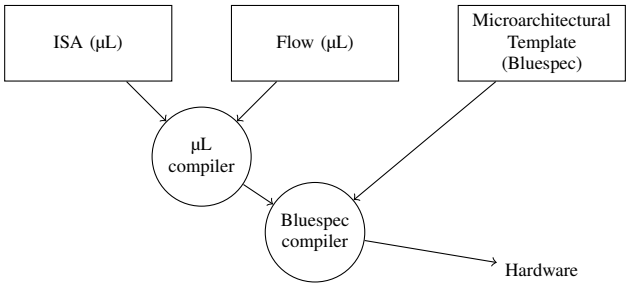


Fig. 3. Overall Toolflow

```
RegID ← defenum [EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI]
let Addr = Bit 32

Inst ← defstruct                                  ▷ Instruction Record
        [ (RegID  , RA)
        , (RegID  , RB)
        , (Bit 32 , IMM)
        , (CCTest , TST)
        ]

State ← defstruct                                 ▷ Architectural State
        [ (Bit 32                 , PC)
        , (Array RegID (Bit 32) , RF)
        , (Array Addr (Bit 8)     , MEM)
        , (CCode                  , CC)
        , (Bool                   , HLT)
        ]
```

Fig. 4. Y86 State

of the code. However, in the presence of placeholder functions, it may not be possible to infer all types. In such cases, type ascriptions can be specified using the @: operator.

*2) Extensible records:* Our language allows record types (structs) to be extended by new fields. Fields are specified by ALLCAPS tags by convention. The +: operator (or rAdd) adds a new field to the record, returning a different record type. The field must not already exist in the record; this is fully ascertained at compile time. The type inference framework is able to infer the return type automatically; the user need not specify it manually. Similarly, −: (or rDel) deletes a field and <: (or rUpd) updates an existing field. We have made the same capability available in Bluespec as a library. Figure 1(b) shows a use of the rAdd function in Bluespec. Extensible records allow us to implement a simple version of subtyping: given a function that expects a record type as the argument, it is legal to pass to it a record type that has more fields than its expects; this property is useful to have for placeholder functions.

*3) Implicit State:* Both the instruction definitions and the flow definition will be functions operating on a state record $\Sigma$. For convenience, the state record is not made explicit in the specification. Instead, its fields can be accessed using special functions **select** and **update**. When the **select**-ed field is of array type, the array can be manipulated into using the **read** and **write** functions. For example, when the state record $\Sigma$ contains a register file at field RF, **select**(RF) selects the entire array, whereas **read**(RF, $r$) returns only the $r$th element of that array.

### B. Y86 ISA

An instruction set architecture (ISA) defines the functional specification of the processor as a set of instructions operating on some programmer-visible state, without prescribing any particular implementation for the state. For example, accesses to the architectural register file may be renamed to a physical register file, so long as the translation is transparent to the programmer.

```
let inc_pc(x) = do                                ▷ Helper function
    pc ← select(PC)
    update(PC, pc + x)

def NOP(inst) = do                                ▷ Instructions
    inc_pc(1)

def SUB(inst) = do
    inc_pc(2)
    a ← read(RF, inst.RA)
    b ← read(RF, inst.RB)
    x ← compute(b − a)
    write(RF, inst.RB, x)
    update(CC, mkcc(CCSUB, b, a, x))

def JXX(inst) = do
    inc_pc(5)
    cc ← select(CC)
    pupdate(PC, test(cc, inst.TST), inst.IMM)

def POP(inst) = do
    inc_pc(2)
    sp ← read(RF, ESP)
    x ← read4(MEM, sp)
    write(RF, ESP, sp + 4)
    write(RF, inst.RA, x)
```

Fig. 5. Y86 Instructions (sample)

The Y86 ISA defines a total of 29 instructions. We found it more convenient to combine the 6 conditional and 1 unconditional jumps into a single JXX instruction, and the 6 conditional and 1 unconditional move into a single RRMOV instruction. These instructions have an additional test argument that specifies the kind of conditional test to perform. (This also reduces the number of entries in the microcode table.) In general, the user can define the instruction in any way that is convenient. It is possible to think of the entire ISA as a single instruction with a giant case statement over the opcode; however our compiler does not perform case-splitting of instructions, hence it is recommended that the user split the case where convenient.

The Y86 ISA specifies a variable-length encoding from 1 to 6 bytes per instruction. To decode an instruction, one must first extract these fields out of the byte-stream into a more convenient record. Figure 4 shows the type definition of the instruction record Inst with fields RA, RB, IMM and TST. The translation from the byte-stream to the instruction record can be specified in Bluespec itself, since it is directly translated to hardware and does not affect the microcode.

Our compiler expects the architectural state to be specified as a record as well (fig. 4). The Y86 ISA specifies eight 32-bit registers organized as an array (RF) indexed by the 3-bit enum type RegID, a flat byte-addressible virtual memory (MEM), a 32-bit program counter (PC), a condition code register (CC) and a bit indicating the processor state (HLT). We ignore interrupts and exceptions in this paper, but section VII-E suggests ways to implement them.

Each instruction is specified as a void function with one argument, the instruction record (fig. 5). The function is allowed to read and write to the architectural state, but nothing else.

The NOP (no-operation) instruction simply increments the program counter (PC) by 1 byte (note that *inc_pc* is a helper function). The SUB instruction is longer: it increments PC by 2 bytes, reads registers at *inst*.RA and *inst*.RB, subtracts the two values, and writes the

```
rule RSTG3;                                              ▷ Memory Stage
    let u3 = stg2.first()
    if (#ld_p(u3) || #st_p(u3))
        let m ← dcache.req(#st_p(u3),                    ▷ request type
                           phyAddr(#addr(u3)),           ▷ address
                           #st_x(u3))                    ▷ store-data
        case(m) matches
            tagged MISS: noAction
            tagged HIT .data:
                let u4 = rAdd (u3, LD, data)
                stg2.deq()
                stg3.enq(u4)
                if (#jmp2_p(u4))
                    jmp2w.wset(#jmp2_x(u4))
    else
        let u4 = rAdd (u3, LD, ?)
        stg2.deq()
        stg3.enq(u4)
        if (#jmp2_p(u4))
            jmp2w.wset(#jmp2_x(u4))
```

Fig. 6. Bluespec code to implement the memory stage, using placeholders and extensible records. Compare lines 20–24 of fig. 7.

result to the register at $inst$.RB. Finally, it calculates condition codes and updates the condition code register (CC). Condition codes are generated and tested by the *uninterpreted functions* $mkcc$ and $test$ respectively. Our compiler does not need to know anything about these functions (other than their type).

The JXX (conditional jump) instruction adds 5 bytes to PC, tests the condition codes using the test specified in $inst$.TST and if it succeeds, sets PC again to $inst$.IMM. An equivalent way to specify this instruction is to perform the test first, and then depending on the result update the PC with either PC + 5 or $inst$.IMM.

The POP instruction reads register ESP and then calls **read4** to load four adjacent bytes from the (byte-addressible) memory MEM. Next, ESP is incremented by 4, and the loaded value is written to $inst$.RA. This instruction is interesting because, although it specifies two writes to the register file, the actual number of writes may be either one or two, depending on whether $inst$.RA is equal to ESP.

### C. Microarchitecture Template

The microarchitecture template is a Bluespec implementation of the microarchitecture with arbitrary combinational logic abstracted away behind placeholder functions. A placeholder is a *pure* function: it has no side-effects, and the returned value is determined entirely by its argument. The Bluespec compiler synthesizes pure functions to combinational logic. In this paper, placeholder functions are indicated using a leading # symbol.

We have implemented a six-stage in-order pipeline with direct-mapped instruction- and data-caches and an always-not-taken branch predictor. Since a lot of these microarchitectural structures are independent of the instruction set, the ability to describe them is entirely that of Bluespec. Since the user can write arbitrary Bluespec, there are practically no constraints on what microarchitectures can be specified.

The fetch stage of the processor pipeline fetches bytes from the cache, and puts them into an instruction buffer. The instruction decoder deciphers the instruction encoding, and translates each instruction into a sequence of one or more micro-instructions (μops), each of which makes a single pass over the hardware. A μop starts off as an extensible record containing a bundle of control-bits from the microcode and all fields of the instruction record. As the μop record travels, it meets resources like register-file ports, multipliers, memory ports etc. What exactly happens at each resource depends on the μop, but is completely specified by the microcode control-bits. The microcode is like the DNA of the μop: it controls how the μop will travel over the hardware, what resources it will activate, how it will be transformed, and how the architectural state will change.

Figure 6 shows the Bluespec code for the memory stage of the pipeline. Note the use of the placeholder functions (*#ld_p*, *#st_p*, *#addr*, *#st_x*, *#jmp2_p*, *#jmp2_x*) in the Bluespec code. These functions will be automatically defined by our compiler. The argument to the placeholder functions is the μop record. Since the placeholder is a pure function, its output is constrained to be determined by the (dynamic) information contained in the μop.

Of course, whether our approach works critically depends on the where these placeholders are inserted. Our compiler assumes that (just like instructions), μops are well-behaved: the microarchitecture preserves all inter-μop dependencies and retires them in the order in which they were created. It is up to the designer to ensure this. Consequently, placeholders cannot appear in code used for handling inter-μop dependencies.

To generate each control bit, our compiler needs to know what resource it will control, *but not the timing of the controlled resource.* Certainly, this information is present in the microarchitecture template, however, extracting it requires the compiler to (at least informally) reason about potentially buggy Bluespec code. This includes understanding microarchitectural structures like caches, branch predictors, data-forwarding buses, floating-point dividers etc. Moreover, inferring how architectural state is physically implemented in hardware is very hard (although user-specified annotations could make this easier). In our memory stage, the compiler would need to infer that the rule of fig. 6 accesses both the memory MEM (via the $dcache.req$ method call) and the program counter PC (via the wire *#jmp2w*) *at most once* per μop.

Since the compiler doesn't need any timing information, we require the user to manually specify the functionality of the template. This specification turns out to be quite short in practice. This approach has an additional advantage: the Bluespec code need not even exist when we run our compiler. This allows the user to study the microcode before actually implementing any hardware.

### D. Flow

Since our compiler assumes that μops are well-behaved and that inter-μop interactions are taken care of correctly, the functionality of a template can be specified by the functionality of an *individual* μop flowing through the pipeline. We call this specification, the *flow*. Like the instruction set, the flow definition is a contract: the designer must ensure that the microarchitecture template implements the specified flow precisely (otherwise the microcode may not work). Note that this contract is much easier to deliver upon than the contract specified by the instruction set.

The flow is defined in a manner similar to the instruction. The main difference is that, in addition to the architectural state, it may have access to so-called *temporary* state—virtual state that is not programmer-visible, but only used for communication between μops of the same instruction (section VII-C).

Figure 7 shows the flow of the μop through our Y86 pipeline. Although it has no timing information in it, to organize the code better, we have shown the names of pipeline stages in comments and by using horizontal markers in the left margin.

The μop starts with all fields of the instruction record ($inst$). The decode stage increments the program counter[2] to whatever value is returned by the *#ilen*($inst$) placeholder, and appends the incremented

```
defflow FLOW(inst) = do                                   ▽ Decode Stage
 2: pc ← select(PC)
 3: npc ← compute(pc + zeroExtend(#ilen(inst) @: Bit 3))
 4: update(PC, npc)                          ▷ increment program counter
 5: u0 ← compute(inst +: (NEXTPC, npc))
                                                    ▽ Register Read Stage
 6: x1 ← pread(RF, #rd1_p(u0), #rd1_r(u0))          ▷ read registers
 7: x2 ← pread(RF, #rd2_p(u0), #rd2_r(u0))
 8: cc ← pselect(CC, #rdcc_p(u0))
 9: p0 ← compute(test(cc, #tst(u0)))
10: u1 ← compute(u0 +: (RD1, x1) +: (RD2, x2)) +: (TEST, p0)
11: pupdate(PC, #jmp1_p(u1), #jmp1_x(u1))                ▷ branch
                                                    ▽ Execute Stage
12: x ← compute(#alu_x(u1) @: Bit 32)          ▷ arithmetic & logic
13: y ← compute(#alu_y(u1) @: Bit 32)
14: z ← compute(if(#alu_k0(u1),
15:                 if(#alu_k1(u1), x + y, x − y),
16:                 if(#alu_k1(u1), x & y, x ^ y)))
17: u2 ← compute(u1 +: (ALU, z))
18: cc ← compute(mkcc(#cc_n(u2), x, y, z))
19: u3 ← compute(u2 +: (MKCC, cc))              ▷ condition codes
                                                    ▽ Memory Stage
20: v ← pread4(MEM, #ld_p(u3), #addr(u3))                ▷ load...
21: pwrite4(MEM, #st_p(u3), #addr(u3), #st_x(u3))        ▷ store...
22: constrain(not(#ld_p(u3) && #st_p(u3)))          ▷ ...but not both
23: u4 ← compute(u3 +: (LD, v))
24: pupdate(PC, #jmp2_p(u4), #jmp2_x(u4))                ▷ branch
25: constrain(not(#jmp1_p(u1) && #jmp2_p(u4)))
                                                    ▽ Writeback Stage
26: pwrite(RF, #wresp_p(u1), ESP, #wresp_x(u4))
27: pwrite(RF, #wr_p(u1), #wr_r(u1), #wr_x(u4))
28: pupdate(CC, #wrcc_p(u1), #cc(u4))
29: pupdate(HLT, #halt(u4), true)                        ▷ halt
```

Fig. 7. Flow Definition (complete)

PC to the μop at a new field NEXTPC (lines 2–5). In the register-read stage, we perform two reads from the register file, and perform a test on the condition code register, appending values to the μop as before (lines 6–10). It is possible to branch (update PC) from here (line 11). The execute stage performs one of four arithmetic/logic operations and calculates the condition codes (lines 12–19). The memory stage potentially accesses memory (MEM) via a load/store port and performs another branch (lines 20–25). Finally, the write-back stage writes back to the architectural state.

There are a few points worth mentioning:

1) The state reads on lines 6–8, 20 are predicated: if the predicate is false, it doesn't matter what value is returned by the read.
2) The flow defines two reads to the register file on lines 6–7. However, the microarchitecture template need not have two *physical* read ports; instead it may provide the illusion of two read ports by taking two cycles over a single physical read port.
3) Like the instruction definitions, the flow definition also has sequential semantics. Thus, the two writes to the register file on lines 26–27 must occur in that sequential order: if both the writes happen to the same register (i.e., if $\#wr\_r(u_1) = \text{ESP}$), the second write will take priority (it appears to clobber the first). The Bluespec template must ensure that this semantics is implemented correctly.
4) The same placeholder can be used in multiple places (provided it has the same argument type). On lines 20–21, *#addr* is used for the address of both the load and the store to memory.
5) The user can specify boolean constraints in the flow. The constraint of line 22 prohibits a μop from specifying both a load and a store. Effectively, the flow is specifying a single port to memory that can do either a load or a store. The constraint of line 25 prohibits a μop from using both "branch ports".
6) Lines 22–24 describe the functionality of fig. 6.

[2]It may shock some readers to notice that PC is incremented in the decode stage, and not the fetch stage. In fact, this is a common pattern in variable-length ISAs because the instruction length is only known after the instruction is decoded. The fetch stage has no idea where an instruction begins and ends.

$$\text{NOP}(inst, \Sigma) = \Sigma <: (\text{PC},\ \Sigma.\text{PC} + 32'h1)$$
$$\text{JXX}(inst, \Sigma) = \Sigma <: (\text{PC},\ \textbf{if}(test(\Sigma.\text{CC},\ inst.\text{TST}),\ inst.\text{IMM},\ \Sigma.\text{PC} + 32'h5))$$
$$\text{FLW}(inst, \Sigma) = \Sigma <: (\text{PC},\ \Sigma.\text{PC} + \text{zeroExtend}_{3\to 32}(\#ilen(inst)))$$

Fig. 8. Some internal terms in canonical form

## V. COMPILATION

The compiler *normalizes* each instruction definition to a canonical form. First, the function defining the instruction is translated into a purely functional representation (term) that makes the state record explicit. This functional term takes two arguments: the instruction record and a state record, and returns the updated state record. Then, the compiler uses a set of predefined rules to repeatedly rewrite the term. The canonical form is reached when the term cannot be rewritten any more. Canonical forms for the NOP and JXX instructions are shown in fig. 8.

Similarly, the compiler also normalizes the flow definition. Since the canonical form for the flow of fig. 7 would be too long to show here, we assume that its description was truncated after line 4 (right after the updating the PC), and show its canonical form in fig. 8.

Though it is very short, this truncated flow specification can implement the NOP instruction. To see this, substitute *#ilen* by a function that always returns $32'h1$, and then rewrite $\text{zeroExtend}_{3\to 32}(3'h1)$ to $32'h1$. This yields exactly the canonical form of the NOP instruction. Thus, $\#ilen(inst) = 32'h1$ is a solution for the NOP instruction. Verifying a solution is easy: substitute the solution into the term, normalize the term, and do a syntactic comparison on the terms.

To generate a solution, the compiler must do this in reverse: it must work backwards reconstructing the sequence of rewrite rules applied during the verifying normalization [12]. This is a computationally hard problem (and undecidable in general).

Put another way, the compiler is really a simple backtracking higher-order theorem prover. As such, it depends heavily on heuristics, and occasionally needs human intervention. Our current heuristics for guiding this theorem prover work well for single μop solutions: more often than not, the prover makes the right choice at the first attempt.

In particular, our compiler covers all Y86 instructions within 10 seconds (fig. 9). On the other hand, when heuristics fail (as they sometimes do while generating multi-μop solutions), the compiler works past the timeout (10 minutes). In this case, the user needs to provide hints to the compiler. Currently our mechanism of providing hints is quite ad hoc, and requires that the user have some understanding of how the compiler works. This is certainly disagreeable, and we are working on a way to provide hints in a more general manner, perhaps as partial solutions to the placeholders.

The output of our compiler is very human-readable (after all placeholders are high-level Bluespec functions). In addition, we have

an verbose mode that unpacks the microcode, so that the fields of the microcode table are labeled by name, making it easier to identify which placeholder they control. This makes code output by the compiler fairly easy to read while debugging.

However, in the presence of bugs in the ISA or flow descriptions, a correct solution may not even exist. In such a case, the compiler often reaches the timeout, and exits without finding a solution. In such a case it is hard to tell exactly *why* a solution could not be found. This can make even minor errors in the input quite hard to debug. However, the user does know *which instructions* caused the failure, and this is often (but not always) sufficient to point out the bug. In our experience, building a processor by adding instructions one by one is often the most productive way to use our toolflow.

## VI. Implementation

The main claim of this work is that this untangling achieved by separating the concern of functionality (architecture) from the concerns of implementing simplifies the description of hardware, without affecting the ability to specify inherently complex hardware like out-of-order execution, trace caches, branch predictors etc. This ability to specify complex hardware comes by design with the programming model exposed by placeholder functions. We are only limited by the capabilities of the underlying hardware description language—in our case, Bluespec SystemVerilog.

We implemented another Y86 processor with the same microarchitecture by hand to compare against the one generated from the described toolflow. The two sets of descriptions are expected to generate exactly the same hardware, assuming Bluespec or downstream synthesis does not do something very unexpected. In fact, the two processor descriptions are exactly the same, except that one of them uses placeholder functions, and the other has pieces of combinational logic "manually inlined" in that place. In particular, the micro-op structure and encoding are the same between the two processors.

### A. Design complexity reduction

Unfortunately, simplicity is hard to measure objectively. We have used lines of code as a proxy metric for simplicity. Obviously, this metric is far from perfect, and should be taken with a pinch of salt.

The Y86 processor written without using our tool takes about 3300 lines of Bluespec SystemVerilog code. (All lines-of-code measurements presented here are correct to 2 significant digits.) However, this includes several reusable library components like multi-ported memories, caches, instruction buffers etc., that are not affected by our programming model and compiler at all. It takes 450 lines of Bluespec code to describe the processor pipeline itself.

When we split the description, the ISA spans 180 lines, the flow specification spans 63 lines, and the Bluespec template reduces to 400 lines, bringing the total to 640 lines. Thus, perhaps surprisingly, the absolute number of lines of code significantly *increase* because of the split: from 450 to 640.

How can we then claim that our toolflow reduces complexity? First of all, in the original description, the user had to split her attention between both the instruction set and the pipeline itself. Second, the ISA is easily re-usable from one generation of microarchitecture to the next, and the cost of specifying it is amortized over time. In addition, the complexity of writing some of these lines of code is significantly lower. 20 of the 450 original lines of code include a microcode table. This microcode table has 17 entries, 30 bits each, or a total of 510 bits, all of which must be manually specified. Here each bit depends on information from both the instruction set and the functionality of the pipeline. The complexity of writing down

| | |
|---|---|
| Typecheck | 0.2 s |
| Normalize | 1.5 s |
| Solver | 4.9 s |
| Microcode | 0.1 s |
| Total | 6.7 s |

(a) Compile times by phase

| | | | |
|---|---|---|---|
| CALL | 0.24 s | LEAVE | 0.35 s |
| FADD | 0.32 s | MRMOV | 0.31 s |
| FAND | 0.31 s | NOP | 0.11 s |
| FSUB | 0.28 s | POP | 0.39 s |
| FXOR | 0.29 s | PUSH | 0.29 s |
| HALT | 0.11 s | RET | 0.43 s |
| IADD | 0.25 s | RMMOV | 0.33 s |
| IRMOV | 0.14 s | RRMOV | 0.42 s |
| JXX | 0.16 s | | |
| | Total | 4.73 s | |

(b) Solver times by instruction

Fig. 9. Measurements

microcode is almost certainly more than just one line of code per 30-bit entry. If we treat the 510 bits as placed on different lines, then the original specification has a total of 940 lines of code compared to our split specification which has 640.

To give an analogy, consider the comparison of Verilog bit-vectors with SystemVerilog structs. In Verilog, the user must figure out exactly the bit-range at which a particular value is packed in a bit-vector. SystemVerilog structs allow the user to specify the structure separately from the description, which can now access fields in the struct using the '.' operator instead of using bit-ranges. Using SystemVerilog structs clearly increases the lines of code, and yet makes the description much easier to write, read and modify.

### B. Compile times

We have improved the instruction solver in our compiler significantly over the work of [2]. Our compiler is now able to compile the Y86 description within 7 seconds, as measured on a 3.33 GHz Intel i7 processor, running on a single thread. Figure 9(a) shows a break-up of the compile times for each phase of the compiler. The equation solver is the most computation-intensive part taking just under 5 seconds. The break-up of the time taken by the solver per instruction is shown in fig. 9(b). It is possible to trivially parallelize this phase by spawning a thread for each instruction.

The older version of the compiler was not able to find a solution for even a single instruction within 30 mins of searching (per instruction). This is counter-intuitive because the NOP instruction should be implementable by even a dumb compiler. The problem here is the complexity of the flow description, in particular, the fact that it specifies two write ports. Writes to distinct elements of an array are commutative; this axiom causes the algorithm implemented in the older compiler to diverge.

## VII. Discussion

### A. RAW hazard handling

As mentioned before, it is up to the user to ensure that inter-µop handling is correct. Our implementation determines inter-µop register dependencies (so-called read-after-write hazards) in the register-read stage, and stalls the pipeline whenever a µop needs to read a register that will be written to by a prior in-flight µop. Thus, the stalling logic must compare the read-register of the µop in the register-read stage with the write-register of every µop after the register-read stage. This strategy only works if the write-register (returned by the placeholder $\#wr\_r$) is known by the time the µop leaves the register-read stage. Thus, $\#wr\_r$ must be constrained to only use data available at the end of register-read stage.

The argument to the placeholder function is generally the latest µop available at that point in the hardware. This gives the compiler the most freedom in assigning values to the placeholders. However, in this case, we must use $\#wr\_r(u_1)$ instead of $\#wr\_r(u_4)$. (The same argument applies to other placeholders on lines 26–28.)

```
def POP(inst) = do
    inc_pc(2)
    sp ← read(RF, ESP)
    write(RF, ESP, sp+4)
    update(TMP, sp)
    ----------
    sp ← select(TMP)
    x ← read4(MEM, sp)
    write(RF, inst.RA, x)
```

(a) Using a temporary

```
def POP(inst) = do
    inc_pc(2)
    sp ← read(RF, ESP)
    write(RF, ESP, sp + 4)
    ----------
    sp ← read(RF, ESP)
    x ← read4(MEM, sp − 4)
    write(RF, inst.RA, x)
```

(b) Without using a temporary

Fig. 10. Partitioning POP instruction, when only one write is available

```
def POP_ESP(inst) = do
    inc_pc(2)
    sp ← read(RF, ESP)
    x ← read4(MEM, sp)
    write(RF, ESP, x)
```

Fig. 11. Specialization of the POP instruction when $inst.\text{RA} = \text{ESP}$

We had missed this aspect in the initial flow description. However, the Bluespec type-checker complained, saying that we had attempted to call $\#wr\_r(u_1)$ in the Bluespec dependency handling logic, but the input argument was missing fields ALU, MKCC and LD (as would be defined in $u_4$). By comparing the arguments of $\#wr\_r$ in the implementation and the flow, we immediately saw the problem.

### B. Microarchitectural costs

Our compiler does not understand microarchitectural costs. For example, the compiler may map the JXX instruction to use either *jmp1* (line 11) or *jmp2* (line 24) to update PC with $inst.\text{IMM}$. However, since *jmp1* is earlier in the pipeline, it has a lower branch mis-speculation penalty, and should be preferred over *jmp2*. Similarly, the compiler may introduce spurious loads (doing so is always legal, but bad for performance), however we haven't seen this occur in practice.

In general, every read/write access to architectural state has a cost that needs to be indicated to the compiler, perhaps through the flow definition. However, this is not enough. For example, when compiling to multiple µops, it is beneficial to lift loads in the earlier µops.

### C. Multiple µops

In the presented example, the compiler is able to map each instruction to a single µop. This is only possible because the flow defines sufficient resources for every instruction. For example, it does two writes to the register file (lines 26–27), as needed by the POP instruction.

Instead, the compiler can break POP to multiple µops. For this, following changes to the flow are necessary: (a) the write to ESP (line 26) needs to be removed; (b) the PC increment (lines 2–5) cannot happen once per µop, but only once per instruction; and (c) (optionally) some temporary state needs to be added to enable data transfer between two µops of the same instruction.

The temporary state is specified alongside the architectural state. Instructions are not permitted to access temporary state, but the flow definition is. The temporary state can be specified as a single register or a register file. (Of course, the Bluespec template must also implement temporary registers as specified by the flow.)

Two possible partitionings of the POP instruction are shown in fig. 10. If the compiler makes a poor choice, the user can manually specify a partitioning into sub-instructions.

Microarchitectures that allow multiple µops per instruction, often have additional restrictions on µops. For example: (a) only the last µop may be allowed to do a branch, since it simplifies flushing µops after a branch mis-speculation; (b) an exception may not occur after a µop that does a store to memory; etc. The user needs to make such restrictions explicit in the flow definition.

### D. Variable number of µops

Currently, our compiler cannot automatically break an instruction into a *variable* number of µops (where the number of µops is determined by a condition). However, the user can manually specialize the instruction. For example, the POP instruction can be specialized to take only one µop for the case when $inst.\text{RA} = \text{ESP}$. (fig. 11). Currently, our compiler cannot do such case-splitting automatically.

### E. Exceptions

Our language does not have any special support for exceptions (although we are considering adding this feature). Instructions that generate exceptions must explicitly set an exception field in the architectural state (while making sure that the architectural state is not modified in any way). The microarchitecture template must ensure that the exception is duely processed. To process an exception, the microarchitecture can introduce a fake EXCP instruction defined to save the program counter on the stack and to jump to the exception handler. Our compiler can generate µops for EXCP in the usual way.

## VIII. Conclusion

We have implemented a Y86 processor with a six-stage in-order pipeline to qualitatively illustrate the complexity reduction from disentangling the description into a functional description and a microarchitectural template. The design is substantially simplified by replacing the contract of implementing the instruction set with the contract of implementing the flow definition.


## Acknowledgments

We would like to thank Prof. Zhiru Zhang for shepherding this paper, several anonymous reviewers for providing valuable comments, and Hari Angepat, Khubaib and Ankit Bansal for helpful discussions. This material is based upon work supported in part by the National Science Foundation under grants 0615352, 0747438, and 0917158.


## References


[1] R. S. Nikhil and K. Czeck, "BSV by Example," 2010.

[2] N. A. Patil, A. Bansal, and D. Chiou, "Enforcing Architectural Contracts in High-level Synthesis," in *DAC*, 2011.

[3] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2nd ed. Addison Wesley, 2010.

[4] A. Bansal, "Generating RTL for microprocessors from architectural and microarchitectural description," Master's thesis, University of Texas at Austin, 2011.

[5] A. Solar-Lezama, "Program Synthesis by Sketching," Ph.D. dissertation, University of California, Berkeley, 2008.

[6] A. Raabe and R. Bodik, "Synthesizing hardware from sketches," in *Wild and crazy ideas, DAC*, 2009.

[7] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011.

[8] P. Mishra and N. Dutt, *Processor description languages.* Morgan Kaufmann, 2008.

[9] R. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro*, 2000.

[10] I. Augé and F. Pétrot, "User-Guided High Level Synthesis," in *High-Level Synthesis*, 2008.

[11] D. Gajski, M. Reshadi, and J. Trajkovic, "No instruction set computer." [Online]. Available: http://www.ics.uci.edu/~nisc

[12] C. Prehofer, *Solving Higher-Order Equations : From Logic To Programming.* Birkhäuser, 1997.